University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Pitt SCI

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - **Assignment 2: ~~Friday 11/4~~ Monday 11/7 @ 11:59 pm**
    - **NO LATE DEADLINE**
  - Lab 7: next Monday 11/7 @ 11:59 pm
  - Homework 8: next Friday @ 11:59 pm
- Live Support Session for Assignment 2
  - Recording and slides on the assignment Canvas page
- Weekly Live QA Session on Piazza
  - Friday 4:30-5:30 pm

Previous lecture

- ADT Graph
  - finding articulation points of a graph
  - Graph compression
  - Graphs with weighted edges
  - Minimum Spanning Tree (MST) problem

This Lecture

- ADT Graph
  - Minimum Spanning Tree (MST) problem
    - Prim's MST algorithm
    - Kruskal's MST algorithm

Muddiest Points

- **Q: Does the articulation point algorithm have a name?**
- It is part of a larger algorithm that finds the biconnected components of an undirected graph by Hopcroft and Tarjan (check Canvas for a link to the original paper from 1971)

Muddiest Points

- Q: Can we get another example of finding the articulation points for a graph?
- Sure!

Muddiest Points

- Q: I do not understand how CSR works at all, can you please re-explain it slower? Thanks
- Q: I don't understand what offsets are and what they represent
- Q: calculating difference array
- Let’s have another example!

Muddiest Points

- **Q: how to calculate the degree of a vertex in constant time with the offset array**
- degree of vertex i = offsets[i+1] – offsets[i]
- Assume that we add an extra entry to offsets:
    - offsets[v] = edges.length

Muddiest Points

- **Q: when will the exam be graded and returned?**
- Almost done; I am 96% through

Muddiest Points

- **Q: why the space of the adjacent linked list is v+2e? why is 2e?**
- **Q: Why is the adjacency lists memory Theta(v+e) and not Theta(v*e)?**
- For each edge in an **undirected** graph, two nodes are added to the adjacency lists; one for each end point
- For each edge in an **undirected** graph, one node is added to the adjacency list of the *from* vertex

Muddiest Points

- **Q:** **I didn't quite get how huffman was related to DFS**
- The codebook construction algorithm in Huffman is an example of DFS traversal of the Huffman Trie

Muddiest Points

- **Q: I don't understand how BFS can verify if parts of a graph are connected or not**
- The graph is connected if and only if a single call to BFS visits all vertices of the graph

Muddiest Points

- **Q: Do acyclic properties only apply to directed graphs?**
- Both directed and undirected graphs can have cycles
- For directed graphs, we sometimes consider cycles and directed cycles

Muddiest Points

- **Q: how does the trivial graph implementation connect the edges with the corresponding vertices?**
- Each edge is stored as a pair of two integers representing the two endpoints

Muddiest Points

- **Q: When is it best to use an adjacency matrix vs an adjacency list?**
- Depends on the application's priority
  - if top priority is space **and** the graph is sparse
    - use adjacency lists
  - if top priority is the isNeighbors(u, v) operation
    - use adjacency matrix
  - otherwise, use adjacency lists
- Adjacency lists store the outward neighbors in directed graphs
  - what if we need to store inward neighbors?

Muddiest Points

- **Q: How do you know the parents of the node when applying BFS to find shortest path**
- We maintain a parents array and update it as we add unseen neighbors to the queue

Muddiest Points

- **Q:** what does epsilon mean
- a small value that is almost zero

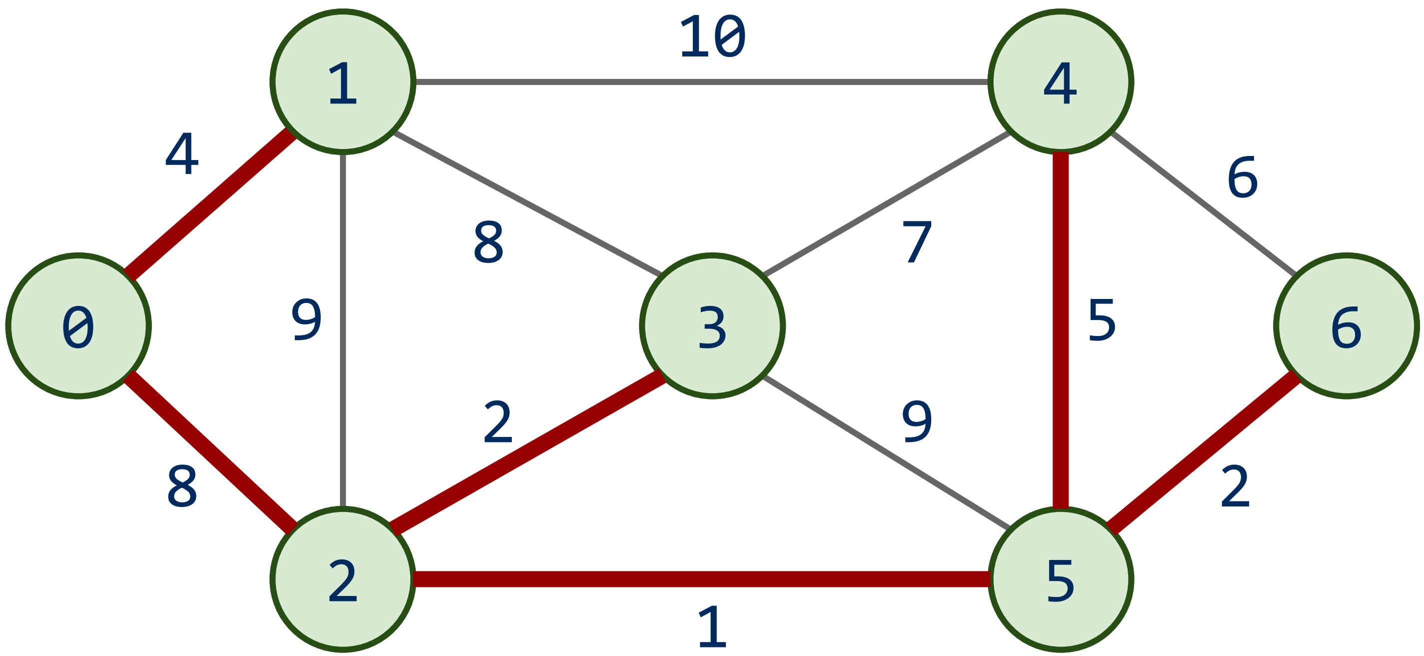
Neighborhood connectivity Problem

- keep a set of neighborhoods connected
  - We can go from any neighborhood to any other
- with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
  - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
  - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
  - Neighborhoods are connected
  - The total cost is minimum

Prim’s algorithm

- Initialize T to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T

Prim’s algorithm



Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - …
- Runtime:
  - $\Sigma_{i\,=\,1\text{ to }v}\,(i * (v - i)) = \Theta(\text{largest term} * \text{number of terms})$
  - number of terms = *v*
  - largest term is $v^2/4$ (when $i=v/2$)
  - Evaluates to $\Theta(v^3)$

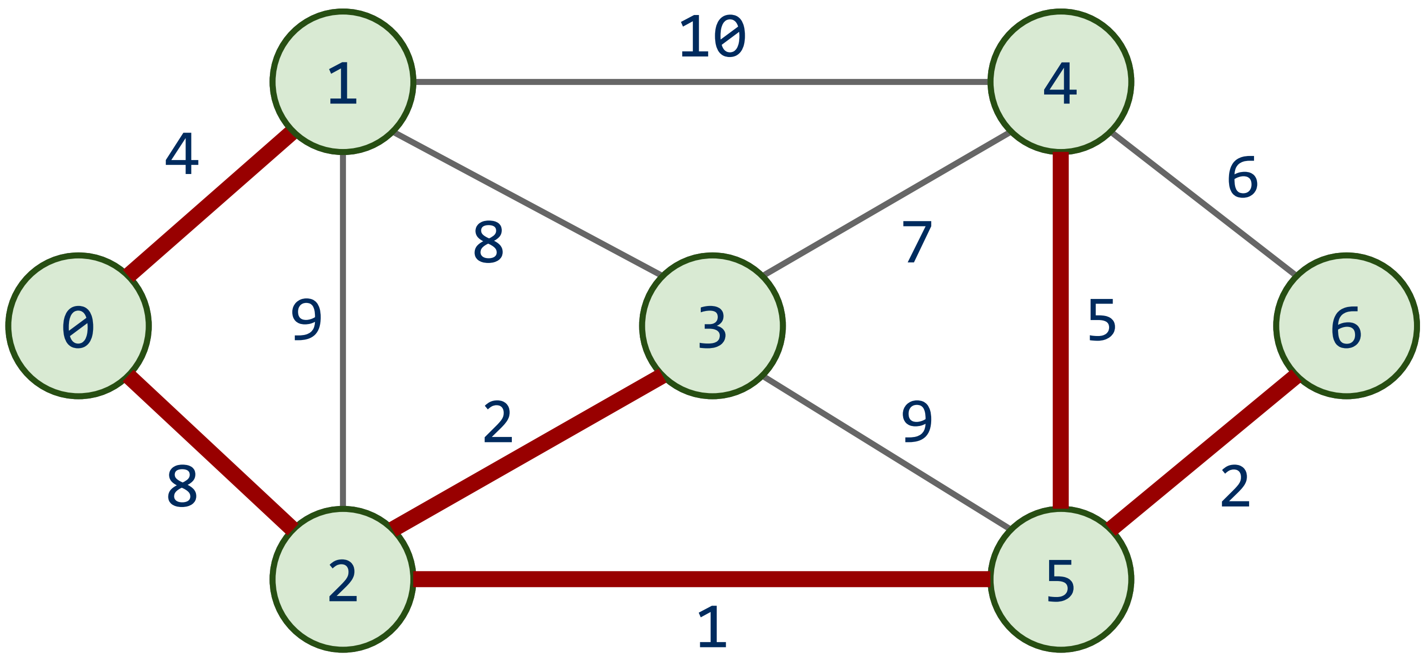
Do we need to look through all remaining edges?

- No! We only need to consider the *best* edge possible for each vertex!
    - The best edge of each vertex can be updated as we add each vertex to T

An enhanced implementation of Prim's Algorithm

- Add start vertex to T
- Search through the neighbors of the added vertex to adjust the parent and best edge arrays as needed
- Search through the best edge array to find the next addition to T
- Repeat until all vertices added to T

Prim’s algorithm



	0	1	2	3	4	5	6
Parent:	--	0	0	2	5	2	5
Best Edge:	0	4	8	2	5	1	2

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
  - Let's assume we use an **adjacency matrix**:
    - Takes Θ(v) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
      - Θ(1)
    - Time to pick next vertex?
      - Θ(v)
    - Total: v*2 Θ(v) = $\Theta(v^2)$

OK, so what's our runtime?

- For every vertex we add to T, we'll need to check all of its neighbors to update their best edges as needed
  - Let's assume we use **adjacency lists**
    - Takes Θ(d) to check the neighbors of a given vertex
    - Time to update parent/best edge arrays?
      - Θ(1)
    - Time to pick next vertex?
      - Θ(v)
    - Total: $v^*\Theta(v + d) = \Theta(v^2)$

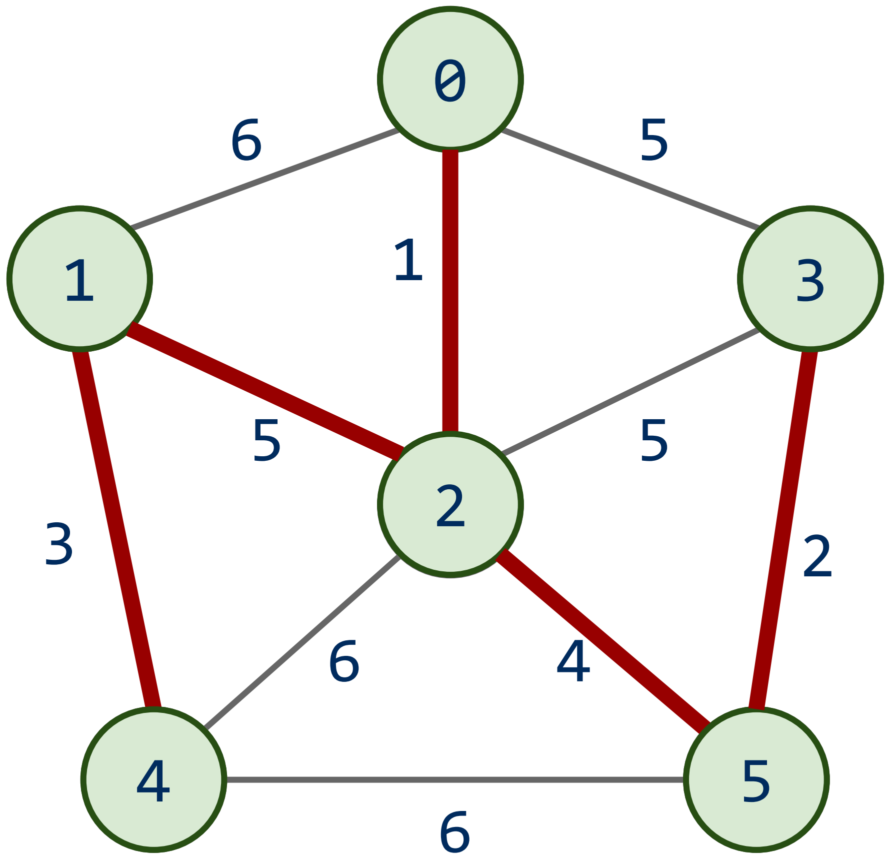
Prim's MST Algorithm

- seen, parent, and BestEdge are arrays of size v
- Initialize seen to false, parent to -1, and BestEdge to infinity
- BestEdge[start] = 0
- for i = 0 to v-1
  - Find a vertex w with seen[w] = false and BestEdge[w] is the minimum over all unseen vertices
  - seen[w] = 1
  - for each neighbor x of w
    - if(BestEdge[x] > edge weight of edge (w, x)
      - BestEdge[x] = edge weight of (w, x)
      - parent[x] = w
- The parent array represents the found MST

What about a faster way to pick the best edge?

- Sounds like a job for a priority queue!
  - Priority queues can remove the min value stored in them in $\Theta(\lg n)$
    - Also $\Theta(\lg n)$ to add to the priority queue
- What does our algorithm look like now?
  - Visit a vertex
  - Add edges coming out of it to a PQ
  - While there are unvisited vertices, pop from the PQ for the next vertex to visit and repeat

Prim's with a priority queue



PQ:

1: (0, 2)

2: (5, 3)

3: (1, 4)

4: (2, 5)

5: (2, 3)

5: (0, 3)

5: (2, 1)

6: (0, 1)

6: (2, 4)

6: (5, 4)

Runtime using a priority queue

- Have to insert all e edges into the priority queue
  - In the worst case, we'll also have to remove all e edges
- So we have:
  - $e * \Theta(\lg e) + e * \Theta(\lg e)$
  - $= \Theta(2 * e \lg e)$
  - $= \Theta(e \lg e)$
- This algorithm is known as *lazy Prim's*

Do we really need to maintain e items in the PQ?

- I suppose we could not be so lazy
- Just like with the best edge array implementation, we only need the best edge for each vertex
  - PQ will need to be indexable to update the best edge
- This is the idea of *eager Prim's*
  - Runtime is $\Theta(e \lg v)$

Eager Prim's Runtime

v insertions : $v \log v$

e updates : $e \log v$

v removals : $v \log v$

$(e+v) \log v = \Theta(e \log v)$

$e \geq (v-1)$

Comparison of Prim's implementations

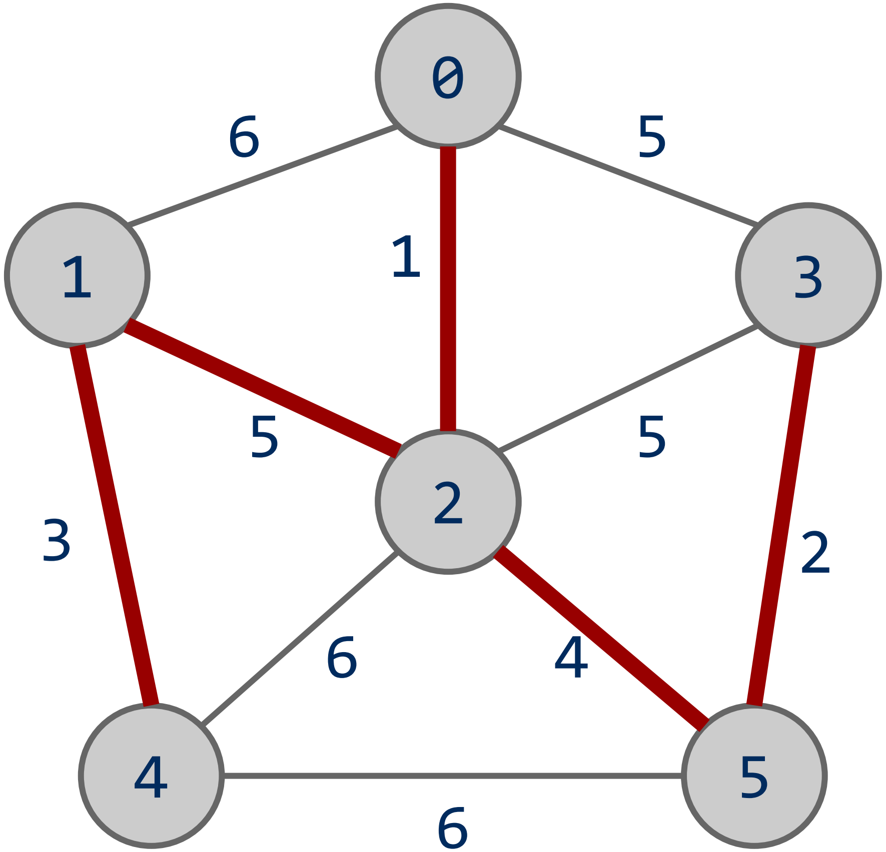
- Parent/Best Edge array Prim's
  - Runtime: $\Theta(v^2)$
  - Space: $\Theta(v)$
- Lazy Prim's
  - Runtime: $\Theta(e \lg e)$
  - Space: $\Theta(e)$
  - Requires a PQ
- Eager Prim's
  - Runtime: $\Theta(e \lg v)$
  - Space: $\Theta(v)$
  - Requires an indexable PQ

How do these compare?

Another MST algorithm

- Kruskal's MST:
  - Insert all edges into a PQ
  - Grab the min edge from the PQ that does not create a cycle in the MST
  - Remove it from the PQ and add it to the MST

Kruskal's example



PQ:

1: (0, 2)
2: (3, 5)
3: (1, 4)
4: (2, 5)
5: (2, 3)
5: (0, 3)
5: (1, 2)
6: (0, 1)
6: (2, 4)
6: (4, 5)

Kruskal’s runtime

- Instead of building up the MST starting from a single vertex, we build it up using edges all over the graph
- How do we efficiently implement cycle detection?

Kruskal's Runtime

e iterations

remove $\quad \log e$

cycle detection $\quad \Theta(v+e)$

DFS/BFS

$e(v+e) = \Theta(e^2)$